

ПРОГРАММИРОВАНИЕ  
ЛЕГКОЕ

# PYTHON ДЛЯ ДЕТЕЙ

САМОУЧИТЕЛЬ ПО ПРОГРАММИРОВАНИЮ

ДЖЕЙСОН БРИГГС



МИФ  
АЕТСТВО



# ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ .....	8
----------------	---

## ЧАСТЬ I

### УЧИМСЯ ПРОГРАММИРОВАТЬ

<b>1. НЕ ВСЕ ЗМЕИ ПРЕСМЫКАЮТСЯ</b> .....	13
Немного о языке .....	14
Установка Python .....	14
Когда Python установлен .....	20
Сохранение Python-программ .....	21
Что мы узнали .....	23
<b>2. ВЫЧИСЛЕНИЯ И ПЕРЕМЕННЫЕ</b> .....	24
Вычисления в Python .....	24
Переменные как ярлыки для данных .....	27
Использование переменных .....	29
Что мы узнали .....	31
<b>3. СТРОКИ, СПИСКИ, КОРТЕЖИ И СЛОВАРИ</b> .....	32
Строки .....	32
Списки мощнее строк .....	39
Кортежи .....	44
Словари в Python — не для поиска слов .....	45
Что мы узнали .....	47
Упражнения .....	48
<b>4. РИСОВАНИЕ С ПОМОЩЬЮ ЧЕРЕПАШКИ</b> .....	49
Использование модуля черепашки .....	49
Что мы узнали .....	56
Упражнения .....	57
<b>5. ЗАДАЕМ ВОПРОСЫ С ПОМОЩЬЮ IF И ELSE</b> .....	58
Конструкция if .....	58
Конструкция if-then-else .....	63
Команды if и elif .....	63
Объединение условий .....	65
Переменные без значения — None .....	65
Разница между строками и числами .....	66
Что мы узнали .....	69
Упражнения .....	70
<b>6. ПРИШЛО ВРЕМЯ ЗАЦИКЛИТЬСЯ</b> .....	71
Использование цикла for .....	71
Цикл while .....	78
Что мы узнали .....	81
Упражнения .....	82
<b>7. ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ КОДА С ПОМОЩЬЮ ФУНКЦИЙ И МОДУЛЕЙ</b> .....	84
Применение функций .....	85
Применение модулей .....	89
Что мы узнали .....	92
Упражнения .....	93

<b>8. КАК ПОЛЬЗОВАТЬСЯ КЛАССАМИ И ОБЪЕКТАМИ</b> .....	95
Разделяем сущности на классы .....	95
Другие полезные свойства объектов и классов .....	104
Инициализация объектов .....	107
Что мы узнали .....	108
Упражнения .....	109
<b>9. ВСТРОЕННЫЕ ФУНКЦИИ PYTHON</b> .....	110
Использование встроенных функций .....	110
Работа с файлами .....	122
Что мы узнали .....	127
Упражнения .....	128
<b>10. ПОЛЕЗНЫЕ МОДУЛИ PYTHON</b> .....	129
Создание копий с помощью модуля <code>copy</code> .....	129
Ключевые слова и модуль <code>keyword</code> .....	132
Генерация случайных чисел с помощью модуля <code>random</code> .....	133
Управление оболочкой с помощью модуля <code>sys</code> .....	135
Работа со временем и модуль <code>time</code> .....	137
Модуль <code>pickle</code> и сохранение информации .....	141
Что мы узнали .....	142
Упражнения .....	143
<b>11. И СНОВА ЧЕРЕПАШЬЯ ГРАФИКА</b> .....	144
Начнем с обычного квадрата .....	144
Рисуем звезды .....	145
Рисуем машину .....	149
Возьмемся за краски .....	150
Функция рисования квадрата .....	153
Рисуем заполненные квадраты .....	155
Рисуем закрашенные звезды .....	156
Что мы узнали .....	158
Упражнения .....	159
<b>12. БОЛЕЕ СОВЕРШЕННАЯ ГРАФИКА С МОДУЛЕМ TKINTER</b> .....	161
Создаем кнопку .....	162
Именованные аргументы .....	164
Создаем холст для рисования .....	165
Рисование линий .....	166
Рисование прямоугольников .....	167
Рисование дуг .....	174
Рисование многоугольников .....	176
Отображение текста .....	177
Вывод изображений .....	179
Создание простой анимации .....	181
Реакция объектов на события .....	183
Для чего еще нужен идентификатор .....	186
Что мы узнали .....	187
Упражнения .....	188

## ЧАСТЬ II

### ПИШЕМ ИГРУ «ПРЫГ-СКОК!»

<b>13. НАША ПЕРВАЯ ИГРА: «ПРЫГ-СКОК!»</b> .....	193
Прыгающий мяч .....	193

Создаем игровой холст .....	194
Создаем класс для мяча .....	195
Добавим движение .....	197
Что мы узнали .....	203
<b>14. ДОДЕЛЫВАЕМ ПЕРВУЮ ИГРУ: «ПРЫГ-СКОК!»</b> .....	204
Создаем ракетку .....	204
Добавим возможность проигрыша .....	210
Что мы узнали .....	214
Упражнения .....	215

### ЧАСТЬ III

#### ПИШЕМ ИГРУ «ЧЕЛОВЕЧЕК СПЕШИТ К ВЫХОДУ»

<b>15. СОЗДАЕМ ГРАФИКУ ДЛЯ ИГРЫ ПРО ЧЕЛОВЕЧКА</b> .....	219
План игры про человечка .....	219
Устанавливаем GIMP .....	220
Создаем изображения для игры .....	221
Что мы узнали .....	228
<b>16. РАЗРАБОТКА ИГРЫ</b> .....	229
Создаем класс игры .....	229
Создаем класс Coords .....	233
Проверка столкновений .....	234
Создаем класс Sprite .....	239
Добавляем платформы .....	240
Что мы узнали .....	244
Упражнения .....	245
<b>17. СОЗДАЕМ ЧЕЛОВЕЧКА</b> .....	246
Инициализация спрайта .....	246
Поворот фигурки вправо и влево .....	249
Прыжок фигурки .....	250
Что мы уже написали .....	251
Что мы узнали .....	252
<b>18. ДОДЕЛЫВАЕМ ИГРУ</b> .....	253
Анимация фигурки .....	253
Проверяем спрайт человечка .....	265
Дверь .....	266
Код игры целиком .....	268
Что мы узнали .....	274
Упражнения .....	275
<b>ПОСЛЕСЛОВИЕ: КУДА ДВИГАТЬСЯ ДАЛЬШЕ</b> .....	276
Игры и программирование графики .....	276
Языки программирования .....	278
Заключение .....	282
<b>ПРИЛОЖЕНИЕ: КЛЮЧЕВЫЕ СЛОВА PYTHON</b> .....	283
<b>ГЛОССАРИЙ</b> .....	297
<b>ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ</b> .....	302

# 8

## КАК ПОЛЬЗОВАТЬСЯ КЛАССАМИ И ОБЪЕКТАМИ

Чем жираф похож на тротуар? Тем, что и жираф, и тротуар — *сущности*, которые в разговорном языке являются *именами существительными*, а в языке Python — *объектами*.

Концепция *объектов* имеет важное значение в мире программирования. Объекты — это способ организации кода в программе, а также способ разделения сложных задач на более простые, что облегчает их решение. (Кстати, в главе 4 нам уже доводилось использовать объект Pen (ручка) для рисования линий.)

Чтобы как следует разобраться, что такое объекты в Python, нужно поговорить об их типах. Начнем с жирафов и тротуаров.

Жираф — это тип (или вид, как говорят биологи) млекопитающих, а млекопитающие, в свою очередь, являются одним из типов животных. Кроме того, жираф — одушевленный объект, поскольку он живой.

Теперь займемся тротуаром. Главное, что можно про него сказать, — он неживой, а значит, относится к неодушевленным объектам. Понятия «млекопитающее», «животное», «одушевленный» и «неодушевленный» — это способы классификации сущностей.



### Разделяем сущности на классы

В языке Python объекты определяются *классами*, благодаря которым объекты можно разделять по смысловым группам. Вот диаграмма классов, которая соответствует нашим рассуждениям о жирафах и тротуарах:



Основной класс — это Сущности. Он имеет два подкласса — Неодушевленные и Одушевленные. Класс Неодушевленные содержит подкласс Тротуары, а класс Одушевленные — подкласс Животные, у которого есть подкласс Млекопитающие, а у него, в свою очередь, подкласс Жирафы.

Классы можно использовать и для организации фрагментов кода в Python-программе. Например, возьмем модуль `turtle`. Все действия, которые может выполнять этот модуль — перемещения черепашки вперед и назад, повороты направо и налево и так далее, — являются функциями класса `Pen`. Объект же является конкретной сущностью, принадлежащей этому классу. Для одного класса можно создать множество объектов, чем мы скоро и займемся.

А пока давайте создадим такой же набор классов, как на нашей диаграмме, начиная сверху. Чтобы определить класс, нужно ввести ключевое слово `class`, а затем указать имя класса. Поскольку класс Сущности (по-английски — `Things`) — наиболее общий из всех, с него и начнем:

**Class** — класс  
**Things** — сущности, вещи

---

```
>>> class Things:
      pass
```

---

Мы дали классу имя `Things` и использовали конструкцию `pass`, обозначающую, что больше никакой информации мы указывать не будем. `Pass` — команда, с помощью которой можно создавать классы или функции, поначалу не программируя их поведение.

Дальше мы добавим остальные классы и зададим некоторые связи между ними.

## Потомки и предки

Когда один класс является частным случаем (подклассом) другого класса, говорят, что первый класс — *потомок*, а второй — *предок*. Один и тот же класс может быть как потомком некоторых классов, так и предком для других классов. На нашей диаграмме класс, находящийся прямо над другим классом, — его предок, а класс под другим классом — его потомок. Например, классы `Inanimate` и `Animate` — потомки класса `Things`, который является их предком.

Для обозначения того, что создаваемый класс является потомком другого класса, нужно указать имя класса-предка в скобках после имени нового класса. Вот так:

`Inanimate` — неодушевленные  
`Animate` — одушевленные  
`Things` — сущности

---

```
>>> class Inanimate(Things):
    pass

>>> class Animate(Things):
    pass
```

---

Здесь мы создали класс `Inanimate`, указав, что его предком является класс `Things`, а затем создали класс `Animate`, также сделав `Things` его предком.

Теперь давайте создадим класс `Sidewalks`, указав его предком класс `Inanimate`:

`Sidewalks` — тротуары

---

```
>>> class Sidewalks(Inanimate):
    pass
```

---

И наконец, аналогичным образом создадим классы `Animals`, `Mammals` и `Giraffes`, не забыв указать в скобках имена классов-предков:

`Animals` — животные  
`Mammals` — млекопитающие  
`Giraffes` — жирафы

---

```
>>> class Animals(Animate):
    pass

>>> class Mammals(Animals):
    pass

>>> class Giraffes(Mammals):
    pass
```

---

## Создаем объекты для классов

Теперь, когда у нас есть набор классов, пора создать принадлежащие им сущности. Предположим, у нас есть жираф по имени Реджинальд. Мы знаем, что он относится к классу `Giraffes`. Как именно описать в программе конкретного жирафа, которого зовут Реджинальд? Будем считать Реджинальда (`reginald`) *объектом* (или, как порой говорят, *экземпляром*) класса `Giraffes`. Чтобы «познакомить» Python с нашим Реджинальдом, нужно написать следующий код:

---

```
>>> reginald = Giraffes()
```

---

Этот код означает: создать объект класса `Giraffes` и присвоить его переменной `reginald`. После имени класса ставятся скобки, как после имени функции. Позже в этой главе мы выясним, как создавать объекты, используя указанные в этих скобках аргументы.

Что может делать объект `reginald`? Пока ничего. Чтобы объекты класса могли решать какие-то задачи, при создании этого класса нужно определить функции, с помощью которых объекты будут делать свое дело. Таким образом, вместо ключевого слова `pass` после определения класса мы можем описать принадлежащие ему функции.

## Определение функций класса

В главе 7 мы уже говорили о функциях как о способе повторного использования кода. Определять функцию, которая принадлежит классу, следует так же, как и обычную функцию, но после определения класса и с отступом. Например, вот обычная функция, не имеющая никакого отношения к классам:

---

```
>>> def this_is_a_normal_function():
    print('Я - обычная функция')
```

---

А вот несколько функций, принадлежащих классу:

---

```
>>> class ThisIsMySillyClass:
    def this_is_a_class_function():
        print('Я - функция класса')
    def this_is_also_a_class_function():
        print('Я тоже функция класса, понятно?')
```

---

**This is a normal function** — это нормальная функция

**This is my silly class** — это мой глупый класс

**This is a class function** — это функция класса

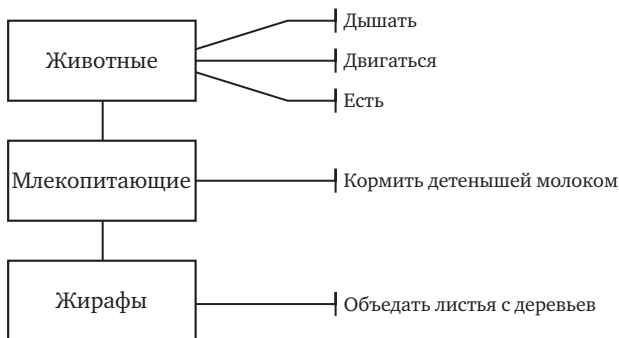
**This is also a class function** — это тоже функция класса



## Используем функции для задания характеристик класса

Рассмотрим классы-потомки класса `Animate`, которым мы дали определение на стр. 97. Можно задать каждому из них *характеристики* (описания, что это за класс и что он может делать). Характеристики — это особенности, присущие всем объектам данного класса, а также объектам классов-потомков.

Например, что общего у всех животных? Они дышат, двигаются и едят. А у млекопитающих? Они вскармливают своих детенышей молоком, а также дышат, двигаются и едят. Жирафы, как известно, объедают листья с верхушек деревьев и, как прочие млекопитающие, кормят детенышей молоком, дышат, двигаются и едят. Если добавить эти характеристики к нашей диаграмме, получится вот что:



Данные характеристики можно представить как действия, или *функции* — то, что объект или класс может делать.

Для добавления функций класса мы используем ключевое слово `def`, поэтому класс `Animals` будет выглядеть примерно так:

```
>>> class Animals(Animate):
    def breathe(self):
        pass
    def move(self):
        pass
    def eat_food(self):
        pass
```

**Breathe** — дышать

**Move** —  
двигаться

**Eat food** — есть  
еду

В первой строке мы определили класс — так, как делали это раньше. Однако следующей строкой вместо `pass` дали определение функции `breathe` с одним аргументом `self`. Аргумент `self` дает функции класса возможность вызывать другие функции этого класса (а также классов-предков). Об использовании этого аргумента мы поговорим немного позже.



Ключевое слово `pass` на следующей строке говорит о том, что мы не собираемся больше ничего сообщать о функции `breathe` — она пока не выполняет никаких действий. Затем идут определения функций `move` и `eat_food`, которые тоже пока ничего не делают. Скоро мы перепишем наши классы и добавим в функции подобающий код. Это довольно типичный подход к разработке программ. Часто программисты создают классы с функциями, которые ничего не делают, чтобы лучше представить,

какими должны быть эти классы, прежде чем переходить к реализации отдельных функций.

Давайте создадим функции и для двух других классов, а именно функцию `feed_young_with_milk` для класса `Mammals` и функцию `eat_leaves_from_trees` для класса `Giraffes`. Каждый класс будет иметь доступ к характеристикам (то есть к функциям) своего предка. Следовательно, ни к чему описывать все особенности класса в нем самом, перегружая его сложным кодом: функции могут располагаться в классах-предках, к которым они логически относятся. Таким образом можно создавать простые и легко читаемые классы.

`Feed young with milk` — кормить детенышей молоком  
`Eat leaves from trees` — есть листья деревьев

---

```
>>> class Mammals(Animals):
    def feed_young_with_milk(self):
        pass

>>> class Giraffes(Mammals):
    def eat_leaves_from_trees(self):
        pass
```

---

## Зачем нужны классы и объекты?

Мы снабдили классы функциями, однако зачем нужны классы и объекты, если можно было написать обычные функции с именами `breathe`, `move`, `eat_food` и так далее?

Прояснить этот вопрос нам поможет жираф Реджинальд, которого мы создали как объект класса `Giraffes`:

---

```
>>> reginald = Giraffes()
```

---

Поскольку `reginald` является объектом, мы можем вызывать функции, определенные в его классе (`Giraffes`) и в его классах-предках. Чтобы вызвать такую функцию для объекта, нужно после имени объекта ввести точку, а затем имя функции. Соответственно, можно дать

Реджинальду команду двигаться или есть, вызывая функции таким образом:

---

```
>>> reginald = Giraffes()
>>> reginald.move()
>>> reginald.eat_leaves_from_trees()
```

---

Теперь предположим, что у Реджинальда есть друг — жираф, которого зовут Гарольд. Давайте создадим еще один объект класса `Giraffes` с именем `harold`:

---

```
>>> harold = Giraffes()
```

---

Поскольку мы используем объекты и классы, при вызове функции `move` можно указать, к какому жирафу этот вызов относится. Например, если мы хотим, чтобы Гарольд передвинулся, а Реджинальд остался на месте, нужно вызвать `move` для объекта `harold`. Вот так:

---

```
>>> harold.move()
```

---

И тогда двигаться будет только Гарольд.

Чтобы это увидеть, слегка изменим код наших классов — вместо `pass` используем в теле каждой функции команду `print`:

---

```
>>> class Animals(Animate):
    def breathe(self):
        print('дышит')
    def move(self):
        print('двигается')
    def eat_food(self):
        print('ест')

>>> class Mammals(Animals):
    def feed_young_with_milk(self):
        print('кормит детенышей молоком')

>>> class Giraffes(Mammals):
    def eat_leaves_from_trees(self):
        print('ест листья')
```

---

Теперь, когда мы создадим объекты `reginald` и `harold` и станем вызывать для них функции, будет видно, что происходит:



---

```
>>> reginald = Giraffes()
>>> harold = Giraffes()
>>> reginald.move()
двигается
>>> harold.eat_leaves_from_trees()
ест листья
```

---

В первых двух строках кода мы создали переменные `reginald` и `harold`, которые являются объектами класса `Giraffes`. Затем вызываем для `reginald` функцию `move`, и в следующей строке Python пишет «двигается». Аналогичным образом вызываем функцию `eat_leaves_from_trees` для `harold`, и Python пишет «ест листья». Если бы это были настоящие жирафы, а не объекты Python, один из них двигался бы, а другой ел.

## Объекты и классы в картинках

Как насчет более наглядного, графического подхода к объектам и классам?

Давайте вернемся к модулю `turtle`, с которым мы упражнялись в главе 4. При вызове функции `turtle.Pen()` Python создает для нас объект класса `Pen` из модуля `turtle` (аналогично тому, как мы создавали объекты `reginald` и `harold` класса `Giraffe` в предыдущем разделе). Так же, как мы создавали двух жирафов, можно создать двух черепашек. Назовем их Эвери (`Avery`) и Кейт (`Kate`):

---

```
>>> import turtle
>>> avery = turtle.Pen()
>>> kate = turtle.Pen()
```

---

Каждый из объектов-черепашек (`avery` и `kate`) принадлежит к классу `Pen`.

Теперь объекты покажут нам свою мощь, ведь после того как объекты-черепашки созданы, можно вызывать функции каждого объекта по отдельности, чтобы черепашки передвигались и рисовали независимо друг от друга.

---

```
>>> avery.forward(50)
>>> avery.right(90)
>>> avery.forward(20)
```

---

Эта последовательность команд переместит Эвери на 50 пикселей вперед, повернет на 90 градусов вправо и переместит вперед на 20 пикселей.

В итоге черепашка будет смотреть вниз. Помните, что вначале черепашки всегда развернуты вправо.

Теперь настала очередь Кейт.

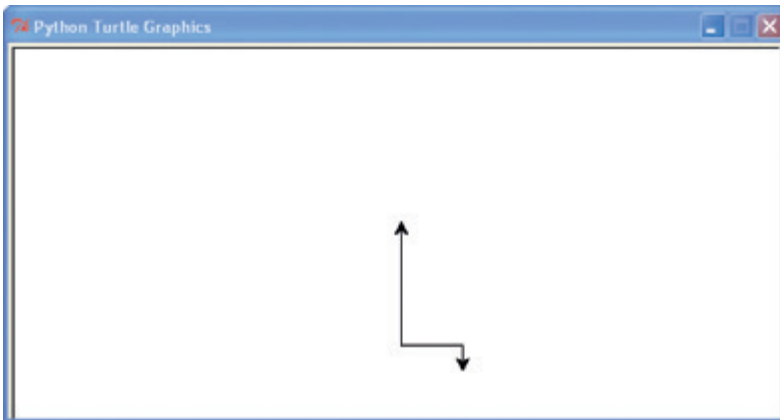
---

```
>>> kate.left(90)
>>> kate.forward(100)
```

---

Мы развернули Кейт на 90 градусов влево и передвинули вперед на 100 пикселей. Она будет смотреть вверх.

У нас получилась линия с двумя стрелочками, направленными в разные стороны, причем каждая стрелочка соответствует одному из объектов-черепашек: Эвери смотрит вниз, а Кейт — вверх.

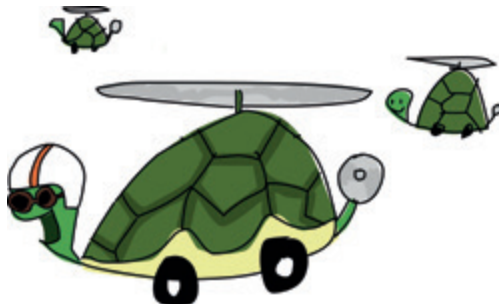


Теперь добавим еще одну черепашку — Джейкоба (Jacob) и переместим его, не беспокоя Эвери и Кейт.

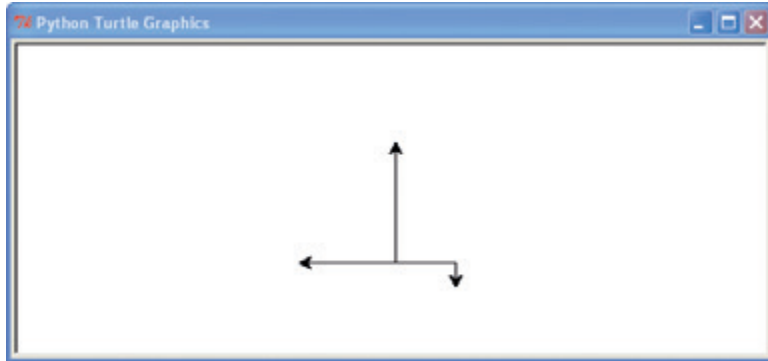
---

```
>>> jacob = turtle.Pen()
>>> jacob.left(180)
>>> jacob.forward(80)
```

---



Создаем новый объект класса `Pen` с именем `jacob`, разворачиваем его на 180 градусов и перемещаем вперед на 80 пикселей. Теперь у нас три черепашки, а картинка выглядит так:



Не забывайте, что каждый раз при создании черепашки с помощью `turtle.Pen()` в программе возникает новый самостоятельный объект. Каждый из таких объектов принадлежит к классу `Pen`, и для каждого из них мы можем вызывать одни и те же функции. Однако действия одной черепашки не связаны с действиями остальных: подобно нашим жирафам (Реджинальду и Гарольду), черепашки Эвери, Кейт и Джейкоб — независимые объекты. Также обратите внимание: если мы создадим новый объект, сохранив его в переменной, где до этого был другой объект, это не означает, что прежний объект тут же исчезнет. Попробуйте сами создать еще одну черепашку Кейт и подвигать ее по холсту.

## Другие полезные свойства объектов и классов

Классы и объекты позволяют удобно группировать функции. Также они помогают представлять программу как набор небольших фрагментов кода.

Представьте действительно большую компьютерную программу — видеоигру или текстовый процессор. Если воспринимать такую программу как неделимое целое, очень сложно будет понять, как она работает, поскольку объем кода слишком велик. Но стоит разбить эту гигантскую программу на части — и вы разберетесь, что делает каждая из частей (разумеется, если знаете язык, на котором программа написана).

Кроме того, если большую программу создавать по частям, становится проще распределять работу между разными программистами. Над наиболее сложными программами (такими как веб-браузер, например) трудятся целые команды программистов по всему миру, одновременно работая над разными частями кода.

Допустим, вам нужно доработать некоторые из созданных в этой главе классов (*Animals*, *Mammals* и *Giraffes*), но времени не хватает, и вы обратились за помощью к друзьям. Тогда работу несложно поделить на части: один человек займется классом *Animals*, другой — классом *Mammals*, а кому-то достанется *Giraffes*.



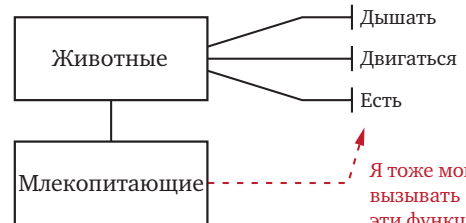
## Унаследованные функции

Наверное, вы уже поняли (если читали внимательно), что программисту, который будет дорабатывать класс *Giraffes*, крупно повезло, ведь в этом классе можно пользоваться функциями классов *Animals* и *Mammals*, которые написаны другими людьми. Класс *Giraffes* наследует функции класса *Mammals*, который, в свою очередь, наследует функции класса *Animals*. Иными словами, создав объект-жирафа, мы получим доступ к функциям класса *Giraffes*, а также классов *Mammals* и *Animals*. Аналогично, создав объект-млекопитающее, мы сможем пользоваться функциями классов *Mammals* и *Animals*.

*Mammals* — млекопитающие

Посмотрите еще раз на связь между классами *Animals*, *Mammals* и *Giraffes*. Класс *Animals* является предком класса *Mammals*, а он, в свою очередь, — предком класса *Giraffes*.

Хоть *reginald* — объект класса *Giraffes*, для него можно вызвать функцию *move* (двигаться), определенную в классе *Animals*, так как функции, определенные в классах-предках, доступны и классам-потомкам:



```
>>> reginald = Giraffes()
>>> reginald.move()
двигается
```

Действительно, все функции, которые мы определили для классов *Animals* и *Mammals*, можно вызвать для объекта *reginald*, поскольку он наследует эти функции:

```
>>> reginald = Giraffes()
>>> reginald.breathe()
дышит
>>> reginald.eat_food()
ест
>>> reginald.feed_young_with_milk()
кормит детенышей молоком
```

## Функции, вызывающие другие функции

Вызывая функцию объекта, мы используем имя переменной, в которой этот объект хранится. Например, так мы вызывали для жирафа Реджинальда функцию `move`:

---

```
>>> reginald.move()
```

---

Если же требуется вызвать функцию `move` из другой функции, определенной в классе `Giraffes`, вместо имени переменной следует использовать аргумент `self`. Таким способом можно вызвать любую функцию класса (или классов-предков) из другой функции класса. Например, добавим в класс `Giraffes` функцию `find_food`:

**Find food** — найти еду

---

```
>>> class Giraffes(Mammals):
    def find_food(self):
        self.move()
        print("Я нашел еду!")
        self.eat_food()
```

---

Мы создали функцию, которая вызывает две другие функции. Это весьма распространенный прием программирования — зачастую, создав полезную функцию, удобно вызывать ее из еще одной функции (и мы займемся этим в главе 13 при создании игры, для которой нам понадобятся более сложные функции).

Добавим в класс `Giraffes` еще несколько функций, использующих `self`:

---

```
>>> class Giraffes(Mammals):
    def find_food(self):
        self.move()
        print("Я нашел еду!")
        self.eat_food()
    def eat_leaves_from_trees(self):
        self.eat_food()
    def dance_a_jig(self):
        self.move()
        self.move()
        self.move()
        self.move()
```

---

**Dance a jig** — танцевать джигу

При создании функций `eat_leaves_from_trees` и `dance_a_jig` мы использовали функции `eat_food` и `move`, принадлежащие



классу-предку `Animals`. Это допустимо, поскольку функции наследуются. Таким образом можно создавать (а затем вызывать для конкретных объектов класса) функции, которые выполняют несколько действий подряд. Смотрите, что произойдет при вызове `dance_a_jig`, — наш жираф передвинется 4 раза (сообщение «двигается» будет напечатано четырежды):

```
>>> reginald = Giraffes()
>>> reginald.dance_a_jig()
двигается
двигается
двигается
двигается
```



## Инициализация объектов

При создании объекта иногда нужно задать какие-нибудь значения (их также называют *свойствами*), которые понадобятся для работы с этим объектом. Подготовка объекта к использованию называется его *инициализацией*.

Например, мы хотим задавать количество пятен на шкуре жирафа при создании каждого объекта класса `Giraffes` (то есть во время инициализации объекта). Для этого нужно создать функцию с именем `__init__` (обратите внимание, что в начале и в конце стоят по два знака подчеркивания). Функция с таким названием имеет особое значение для классов — позволяет задать свойства объекта при его создании, поскольку для каждого нового объекта она вызывается автоматически. Вот как пользоваться этой функцией:

```
>>> class Giraffes:
    def __init__(self, spots):
        self.giraffe_spots = spots
```

Введя `def __init__(self, spots):`, мы создали функцию инициализации, которая принимает два аргумента — `self` и `spots`. Как и остальные функции классов, функция инициализации должна принимать значение `self` первым аргументом. В теле функции строка `self.giraffe_spots = spots` присваивает значение аргумента `spots` свойству объекта `giraffe_spots` (свойство — это принадлежащая объекту переменная). Таким образом, эта строка означает «Взять значение аргумента `spots` и поместить его в переменную объекта (свойство объекта) с именем `giraffe_spots`». Как и в случае вызова одной функции класса из другой, для доступа к свойствам используется аргумент `self`.

**Init** — от initialize — установить начальное состояние  
**Spots** — пятна

Теперь испытаем функцию инициализации, создав еще пару объектов — тоже жирафов (назовем их Освальд и Гертруда) — и проверив, сколько у них пятен:

---

```
>>> ozwald = Giraffes(100)
>>> gertrude = Giraffes(150)
>>> print(ozwald.giraffe_spots)
100
>>> print(gertrude.giraffe_spots)
150
```

---

Мы создали объект класса `Giraffes`, указав в скобках значение 100. В результате была вызвана функция `__init__` с аргументом `spots`, равным 100. Затем мы создали еще один объект класса `Giraffes`, на этот раз указав для аргумента `spots` значение 150. И наконец, напечатав значения свойств `giraffe_spots` для обоих объектов, мы получили 100 и 150. Ура, работает!

Запомните, что после создания объекта (такого как объект `ozwald` в этом примере) для обращения к его функциям и свойствам используются точка и имя переменной (например, `ozwald.giraffe_spots`). Однако при написании функций класса обращаться к этим же свойствам и функциям нужно через аргумент `self` (`self.giraffe_spots`).

## Что мы узнали

В этой главе мы использовали классы для классификации сущностей, а также создавали объекты (экземпляры) этих классов. Узнали, что класс-потомок наследует функции классов-предков и, даже если объекты принадлежат одному классу, они могут отличаться друг от друга (например, у разных объектов — жирафов — может быть разное количество пятен). Мы научились вызывать для объекта функции, определенные в его классе, и сохранять значения в свойствах объектов. И наконец, мы выяснили, как с помощью аргумента `self` получить доступ к функциям и свойствам класса из других его функций. Все это — фундаментальные понятия языка Python, и вы еще не раз встретитесь с ними на страницах этой книги.

## Упражнения

Чтобы как следует разобраться в классах и объектах, нужно начать ими пользоваться. Постарайтесь выполнить эти задания, не подглядывая в подсказки.

### #1. Жирафий танец

Добавьте в класс `Giraffes` функции, при вызове которых жираф представлял бы правую или левую ногу вперед либо назад. Назвать их можно так: `left_foot_forward`, `left_foot_back`, `right_foot_forward` и `right_foot_back`. Функция, которая ставит левую ногу жирафа вперед, может выглядеть примерно так:

```
>>> def left_foot_forward(self):  
        print('левая нога вперед')
```

`Left foot forward` —  
левая нога  
вперед

`Left foot back` —  
левая нога назад

`Right foot  
forward` — правая  
нога вперед

`Right foot back` —  
правая нога  
назад

`Dance` —  
танцевать

Теперь создайте функцию `dance`, которая научит Реджинальда танцевать (вызывая четыре только что созданные функции для передвижения ног). В результате должен получиться несложный танец:

```
>>> reginald = Giraffes()  
>>> reginald.dance()  
левая нога вперед  
левая нога сзади  
правая нога вперед  
правая нога сзади  
левая нога сзади  
правая нога сзади  
правая нога вперед  
левая нога вперед
```

### #2. Черепахи вилы

При помощи четырех черепашек (объектов класса `Pen`) изобразите вилы, как на этой картинке (длину линий выберите на свой вкус). Не забудьте первым делом импортировать модуль `turtle`!

